Dynamically Swappable Hardware Design in Partially Reconfigurable Systems

Chun-Hsian Huang, Kai-Jung Shih, Chao-Sheng Lin, Shih-Shiue Chang, and Pao-Ann Hsiung[†]
Department of Computer Science and Information Engineering
National Chung Cheng University, Chiayi, Taiwan−621, ROC
[†]E-mail: hpa@computer.org

Abstract— **In this work, we propose two wrapper designs for arbitrary digital hardware circuit designs such that they can be enhanced with the capability for dynamic swapping controlled by software. A hardware design with either of the proposed wrappers can thus be swapped out of the partially reconfigurable logic at runtime in some intermediate state of computation and then swapped in when required to continue from that state. The context data is saved to a buffer in the wrapper at interruptible states, and then the wrapper takes care of saving the hardware context to communication memory through a peripheral bus, and later restoring the hardware context after the design is swapped in. The overheads of the hardware standardization and the wrapper in terms of additional reconfigurable logic resources and the time for context switching are small and generally acceptable. With the capability for dynamic swapping, high priority hardware tasks can interrupt low priority tasks in real-time embedded systems so that the utilization of hardware space per unit time is increased.**

I. Introduction

Owing to rapid technology breakthroughs, FPGAs such as Xilinx Virtex II and 4 can now be partially reconfigured at run-time, which means a hardware circuit can be changed while some other hardware circuits are still functioning. Partially reconfigurable systems enable more applications to be accelerated in hardware, and thus reduces overall system execution time [7]. This technology can now be used in real-time embedded systems for switching from a low priority hardware task to a high priority hardware task. However, hardware circuits are generally not designed to be switched or swapped in and out, as a result of which partial reconfigurability either becomes useless or incur significant time overhead. In this work, we try to bridge this gap by proposing two kinds of generic wrapper designs for hardware IPs such that they can be enhanced with the capability for dynamic swapping. The dynamically swappable design must solve several issues related to switching hardware IPs, including (1) when must a hardware design be interrupted for switching? (2) how and where must we save the context of a hardware design? (3) how must we restore the context of a hardware design? (4) how to make the wrapper design small, efficient, and generic? (5) how must a hardware IP be modified so that it can interact with the wrapper.

For ease of explanation, henceforth we call a running hardware circuit as a hardware task. To swap out a hardware task so that it can be swapped in later, one needs to save its execution context so that it can be restored in the future. However, unlike software processes, hardware tasks cannot be interrupted in each and every state of computation. Hence, a hardware task should be allowed to run until the next interruptible state, which is function-specific. The context of a hardware task is also function-specific. Nevertheless, we can use the memento design pattern [1] from software engineering, which states that the context of a task can be stored outside in a memento and then restored when the task is reloaded. To restore a saved context, the context data needs to be preloaded into the wrapper, which then loads the data to the registers in the hardware task. The wrapper architectures are generic so that any digital hardware IP that has been automatically standardized, can be interfaced with it for dynamic swapping. The wrappers receive the software request signals through a task interface and then drive the appropriate signals to prepare the hardware task for swapping. However, the original hardware IP also needs to be enhanced so that it can interface with the wrapper, which we call *standardization*. The detailed descriptions of the wrappers and the hardware task modification are given in Section III.

This work contributes to the state-of-the-art in the following ways.

1) *Generic Wrapper Designs*: These proposed generic wrapper designs can be used to interface with any standardized hardware IP, thus they are reusable and reduce IP development effort significantly. We propose two different wrapper designs to get higher performance and using lesser resources under different conditions.
2) *Swappable Hardware IP*: A hardware IP needs only to be enhanced slightly and interfaced with the wrappers for dynamic swapping.
3) *Better Real-Time Response*: Compared to state-of-the-art methods, our method saves hundreds of microseconds, which give better real-time response during the hardware-software scheduling in an operating system for reconfigurable systems.

This paper is organized as follows: Section II discusses related research work and compares them with our architecture. The details of the dynamically swappable architecture are given in III. We use some applications to demonstrate the validity and genericity of the architecture in IV. Finally, conclusions and future work are described in V.

II. Related Work

Dynamic switching or relocation of hardware designs in reconfigurable logic has been investigated in several previous work, which can be categorized into two classes, namely *reconfiguration-based* [2], [4] and *design-based* [3], [6]. The reconfiguration-based method requires readback support from the reconfigurable logic and deep knowledge of the reconfiguration process for tasks such as state extraction from the readback stream and manipulation of the bitstreams for context restoring. As a result, this method becomes technology dependent and thus non-portable. Another drawback is the poor data efficiency because only a maximum of about 8% of the readback data actually contains state information but all data must be readback to extract the state [2]. The design-based method is self-sufficient because all context switching tasks are taken care of by the hardware design itself through a switching circuitry and registers can be read out or preloaded by the switching circuitry.

Our proposed method for dynamic hardware switching falls into the design-based category, however, we try to eliminate some of the deficiencies of this method, while retaining the advantages. Our method proposes two generic configurable wrappers with a standard interface such that any digital hardware design following the standard can be transformed automatically into dynamically switchable by interfacing with one of the proposed wrappers. Using our proposed method, we have thus not only retained the advantages of data efficiency and technology independence of design-based methods, but also acquired the advantage of reconfiguration-based methods, that is, minimal user design effort for making a hardware IP dynamically reconfigurable. Abstraction of tasks from its hardware or software implementation requires an operating system that can manage both software and hardware tasks. Such an *Operating System for Reconfigurable Systems* (OS4RS) [5] supports the dynamic switching of hardware tasks.

III. Dynamically Swappable Design

A dynamically reconfigurable system consists of a microprocessor attached to a system bus with a communication memory, and a dynamically reconfigurable logic component such as FPGA, within which hardware tasks can be configured and attached to a peripheral bus which in turn is connected through a bridge with the system bus. Each hardware task consists of a hardware IP, a wrapper, and a task interface. The hardware IP is an application-specific function such as a DCT or an H.264 video decoder. In this work, two wrappers are proposed for dynamically swappable design. They control the whole swap-out and swap-in processes of the hardware task. The task interface is an interface to a peripheral bus for data transfers in a hardware task. The task interface acts as a bus interface of the hardware task and is responsible for normal data transfer operations through the control, read, and write interfaces and for swapping and reconfiguration operations through the swap interface.

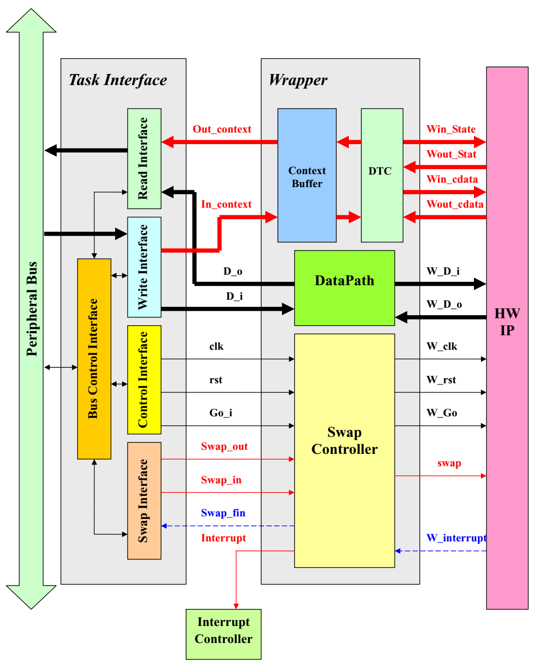

Fig. 1. Wrapper architecture and interfaces

A. Generic Wrapper Designs

Two generic wrapper architectures are proposed for controlling the swapping of a hardware circuit into and out from a reconfigurable logic such that all swap circuitry is implemented within the wrappers with minimal changes to the hardware IP itself. As shown in Figure 1, the wrapper architectures consist of a *context buffer* (CB) to store context data, a data path for data transfer, a *swap controller* (SC) to manage the swap-out and swap-in activities, and some optional *data transformation components* (DTCs) for (un)packing data types. A generic wrapper architecture interfaces with a hardware IP and a standard task interface that connects with a peripheral bus. The difference between the two wrappers lies in the swap-out mechanism and the hardware state in which the IP is swapped out. The *Last Interruptible State Swap* (LISS) wrapper stores the IP context at each interruptible state, thus the IP can be swapped out from the last interruptible state whenever there is a swap request. The *Next Interruptible State Swap* (NISS) wrapper requires the IP to execute until the next interruptible state, store the context, and then swap out. In Figure 1, the LISS wrapper does not include the `W_interrupt` and `Swap_fin` signals, while the NISS wrapper does (signals are highlighted using dotted arrows). The different swap-out processes and the same swap-in process are described as follows.

1) LISS Wrapper Swap-out: At every interruptible state, the context of hardware IP is stored in a *Context Buffer* using the `Wout_State` and `Wout_cdata` signals. When there is a `Swap_out` request from the OS4RS for some hardware task, the wrapper sends an `Interrupt` signal to the microprocessor to notify the OS4RS that (1) the context data stored in the context buffer can be read and saved into the *communication memory*, and (2) the resources (columns) can be deallocated and reused (reconfigured). The swap-out process is thus completed. This wrapper can be used for hardware circuits whose context data size is less than that of the context buffer, as a result of which all context data can be stored in the context buffer using a single data transfer.

2) NISS Wrapper Swap-out: When there is a `Swap_out` request from the OS4RS for some hardware task, the *Swap Controller* in the wrapper sends a `swap` signal (asserted high), to the hardware IP, which starts the whole swap_out process. However, the hardware IP might be in an unswappable state, thus execution is allowed to continue until the next swappable state is reached. At a swappable state, the context of hardware IP, including current state information and selected register data, is stored in a *Context Buffer* in the wrapper using the `Wout_State` and `Wout_cdata` signals. The hardware IP then sends an acknowledgment `W_interrupt` to the wrapper that the swap-out process can continue. The wrapper sends an `Interrupt` signal to the microprocessor to notify the OS4RS that the context data stored in the context buffer can be read and saved into the *communication memory*. This wrapper can be used when the context data size is larger than that of the context buffer by repeating the process of storing into buffer, interrupting microprocessor, and reading into memory. Finally when all context data have been stored into the communication memory, the wrapper sends a `Swap_fin` signal to the task interface, thus notifying the OS4RS that the resources occupied by the IP can be deallocated and reused. The swap-out process is thus completed.

3) Swap-in: When a hardware task is scheduled to execute, the OS4RS configures the corresponding hardware IP with wrapper and task interface into the reconfigurable logic using the *Internal Configuration Access Port* (ICAP), reloads the context data from the communication memory to the context buffer in the wrapper, and sends a `Swap_in` request to the swap controller, which then starts to copy the context data from the buffer to the corresponding registers in the IP using `Win_State` and `Win_cdata`. After all context data are restored, the swap controller sends a `swap` signal (asserted low) to the hardware IP, which then continues from the state in which it was swapped out.

It must be noted here that context data might be of different sizes for different hardware IPs, so data packing and unpacking are performed using the *Data Transformation Component* (DTC) within the wrapper. For the standardized GCD IP example given in Figure 2, there are two 8-bit `X_Wout_cdata` and `Y_Wout_cdata` signals from the IP, which are packed by the DTC in the wrapper into a 32-bit `Out_context` signal for storing into communication memory through the peripheral bus. The other signals in Figure 1 are used for normal IP execution.

B. Standardizing Hardware IP

A sequential circuit is controlled by a *finite state machine* (FSM) through the present and next state registers. Generally, a hardware IP has one or more data registers for storing intermediate results of computation. The collection of the state registers and data registers constitutes the *task context*. A state is said to be *interruptible* if the hardware task can resume execution from that state after restoring the task context, either partially or fully. For the FSM of a GCD IP example given in Figure 2, only the INIT, RLD, and CMP states are interruptible because the comparator results are not saved and hence we cannot resume from the NEG, EQ, and POS states. Making a state interruptible brings no benefit to the task itself, however it can shorten the overall system schedule. A hardware IP is standardized automatically by making the context registers accessible to the wrapper and by enhancing the FSM controller such that the IP can be stalled or the context data can be saved to the context buffer in the wrapper at each interruptible state.

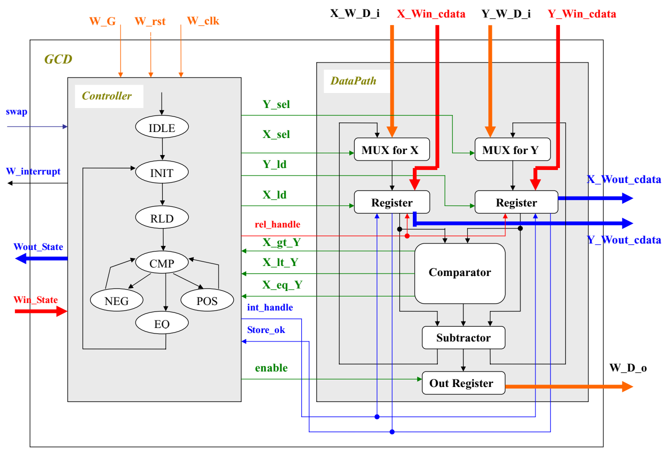

Fig. 2. Swappable GCD circuit architecture

IV. Experiments

We performed all our experiments on the Xilinx Virtex II Pro XC2VP20-FF896 FPGA chip that includes 18,560 LUTs and 18,560 Flip-Flops. The FPGA has a configuration clock of 50 MHz and a full bitstream size of 1,026,820 bytes. All swappable hardware tasks are connected to a 32-bit CoreConnect OPB bus operating at 133 MHz. The OS4RS running on the PowerPC was based on an in-house extension of the Linux OS. We used the Synplify synthesis tool and the ModelSim simulator to verify the correctness of the wrappers and the modified hardware IP designs. We compared the original hardware IP designs with the new swappable ones for each of the six examples, including *Traffic Light Controller* (TLC), *Multiple Light Controller* (MLC), 8-bit Greatest Common Divisor (GCD), 32-bit GCD, *Data Encryption Standard* (DES) encrypter, and *Discrete Cosine Transform* (DCT). We compared the two wrappers for the first three examples because their context data size is less than the context buffer size (32-bits). We made the last three example IPs swappable using the NISS wrapper.

The resource overhead required for making a hardware IP swappable includes the extra resources required to make the

TABLE I

SYNTHESIS RESULTS AND RESOURCE OVERHEADS

HW	Wrapper			D_C	FF			LUT		
	V	FF	LUT	(bits)	IP	SIP	$+\%$	IP	SIP	$+\%$
T	L	3	1	3	6	10	66	24	39	62
	N	3	1			10	66		43	79
M	L	3	1	3	13	17	30	63	77	22
	N	3	1			17	30		77	22
G_8	L	3	1	19	23	23	0	80	114	42
	N	2	1			27	17		122	52
G_{32}	N	5	8	67	71	73	2	270	360	33
DES	N	7	61	836	137	207	51	589	603	2
DCT	N	8	73	1030	1573	2094	33	1339	1152	-13

V: Version, D_C: Context data size, T: TLC, M: MLC, G_8: 8-bit GCD, G_{32}: 32-bit GCD, L: LISS wrapper, N: NISS wrapper, IP: IP resource usage, SIP: Swappable IP resource usage, $+\%$: % of overheads in SIP compared to IP

TABLE II

TIME OVERHEADS FOR SWAP-OUT AND SWAP-IN

HW	W	T_E	Swap-Out			Swap-In			T_R
			T_B	T_P	T'_{SO}	T_B	T_P	T'_{SI}	
T	L	17	2	3	39	2	3	39	42,025
	N		3	3	64	2	3	50	46,336
M	L	33	2	3	50	2	3	50	83,243
	N		3	3	64	2	3	50	83,243
G_8	L	511	2	3	38	2	3	38	122,844
	N		4	3	46	2	3	38	131,465
G_{32}	N	1671	11	9	157	5	9	108	387,931
DES	N	1424	84	81	962	55	81	840	649,784
DCT	N	71552	100	99	1600	66	99	1309	1,481,586

T_E: execution time (in IP clock cycles),
$T_B = \frac{D_B}{R_T} + \frac{D_B}{R_B}$ (in IP clock cycles), $T_P = T_A + \frac{D_B}{R_P}$ (in bus cycles)
$T'_{SO} = T_{SO} - T_R$ (in ns), $T'_{SI} = T_{SI} - T_R$ (in ns)

context registers and the current state register visible and the resources for the wrapper design. Our synthesis results and comparisons are given in Table I. For task G_8, the FF and LUT overheads are 0% and 42% for LISS, and 17% and 52% for NISS, respectively. The negative LUT overhead of the swappable DCT results from optimization of FPGA resource usage. We can observe that the overheads in making the IPs swappable for interfacing with the LISS wrapper are smaller than that for interfacing with the NISS wrapper. This is due to the lesser number of signals in LISS wrapper and the more complex circuitry in NISS wrapper for transferring context data of sizes greater than that of context buffer. The larger the amount of context data, the more FPGA resources are needed for the wrapper DTC, and the more I/O primitives to interact with the swappable hardware IP.

As shown in Table II, the time overheads in swapping out and swapping in for all the examples consume only a few cycles and are in the order of nanoseconds. Given context data of D_C-bits, context buffer of D_B-bits, data transformation rate of R_T bits/cycle, buffer data load rate of R_B bits/cycle, peripheral bus data transfer rate of R_P bits/cycle, peripheral bus access time of T_A cycles, transition time of T_I cycles to go to an interruptible state (T_I is 0 for LISS), and reconfiguration time of T_R cycles, the swap-out and swap-in processes require time T_{SO} and T_{SI}, respectively, as shown in Equation (1). However, all other times in Equation (1) are only a few cycles, in the ns order of magnitude.

$$\begin{aligned} T_{SO} &= T_I + \left\lceil \frac{D_C}{D_B} \right\rceil \times \left(\frac{D_B}{R_T} + \frac{D_B}{R_B} + T_A + \frac{D_B}{R_P} \right) + T_R \\ T_{SI} &= T_R + \left\lceil \frac{D_C}{D_B} \right\rceil \times \left(\frac{D_B}{R_T} + \frac{D_B}{R_B} + T_A + \frac{D_B}{R_P} \right) \end{aligned} \quad (1)$$

From Table II, we can observe that not only is swapping faster with the LISS wrapper, but its simpler circuitry also requires lesser reconfiguration time T_R, compared to NISS. However, as mentioned before LISS wrappers can only be used when the IP context size is not greater than that of the context buffer size. We can thus conclude that if the IP context size is not greater than buffer context size then LISS wrapper is recommended, while the NISS wrapper is required otherwise. We assume typical OPB read and write data transfers for swap-out and swap-in, respectively, hence each of them needs 3 bus cycles for a single 32-bit data transfer. In contrast, the reconfiguration-based methods [2], not only require a reconfiguration time of $190 \sim 6793\mu$s, but also a readback time of $33.6 \sim 129.6\mu$s, while we require only a few tens or hundreds of nanoseconds. We are thus saving tens or hundreds of microseconds, which is important for hard real-time systems.

V. CONCLUSIONS

We have proposed a method for the automatic modification and enhancement of a hardware IP such that it becomes dynamically swappable under the control of an operating system for reconfigurable systems. We have designed two wrappers and analyzed the conditions for using the wrappers. We have still proposed how the hardware IP can be minimally changed by only making the state and context registers visible. The proposed method and architectures were implemented and verified. Our experiment results show that the resource and time overheads of making an IP swappable are almost negligible compared to the amount of reconfigurable resources available and the configuration time of the IP, respectively.

REFERENCES

[1] R. Gamma, R. Helm, R. Johnson, and J. Vissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1994.

[2] H. Kalte and M. Porrmann. Context saving and restoring for multitasking in reconfigurable systems. In *Proc. of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 223–228, August 2005.

[3] J.-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip. In *Proc. of the Design Automation and Test in Europe (DATE)*, pages 986–991, March 2003.

[4] H. Simmler, L. Levinson, and R. Männer. Multitasking on FPGA coprocessors. In *Proc. of the 10th International Workshop on Field Programmable Gate Arrays (FPL)*, pages 121–130, 2000.

[5] C. Steiger, H. Walderand, and M. Platzners. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Transactions on Computers*, 53(11):1392–1407, November 2004.

[6] M. Ullmann, M. Hübner, B. Grimm, and J. Becker. An FPGA run-time system for dynamical on-demand reconfiguration. In *Proc. of the 11th Reconfigurable Architectures Workshop (RAW)*, page 135, 2004.

[7] Xilinx. XAPP290 – two flows for partial reconfiguration module-based or difference-based.